DPUP :
A DISTRIBUTED PROCESSING
UTILITIES PACKAGE

Timothy J. Gardner
Isabelle M. Gerard
Carla R. Mowers
Evi Nemeth
Robert B. Schnabel

CS-CU-337-86 July 1986

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

Research supported by AFOSR grant AFOSR-85-0251, ARO contract DAAG 29-84-K-0140, NSF cooperative agreement DCR-8420944, and Sun Microsystems, Inc.

Abstract

DPUP is a library of utilities that support distributed concurrent computing on a local area network of computers. The library is built upon the interprocess communication facilities in Berkeley Unix 4.2bsd. Thus it will run on any network, connected by an Ethernet, where each computer runs a version of the Unix operating system that supports the Berkeley Unix interprocess communication facilities. DPUP supports two models of distributed concurrent computation, a master-slave model based upon stream sockets, and a broadcast model based upon datagram sockets. With each model, facilities for creating and terminating remote processes, establishing communications between them, and sending and receiving data between these processes are provided. This paper describes the facilities provided in DPUP and gives examples of their use.

1. Introduction

This paper describes a library of utilities that support distributed concurrent computing on a local area network of computers. The library is built upon the interprocess communication facilities in Berkeley Unix 4.2bsd. Thus it will run on any network of computers where each machine runs a version of the Unix operating system that supports the Berkeley Unix interprocess communication facilities. The library is called DPUP for Distributed Processing Utilities Package. It is written in C and can be used by C or FORTRAN applications programs.

The purpose of DPUP is to make it easier to use a local area network of computers as a loosely coupled multiprocessor. It is clear that networks of computers, especially computer workstations, are becoming an increasingly common computing environment in industry and research laboratories. It is inevitable that some users of these networks will want to utilize a number of computers simultaneously, as a loosely coupled multiprocessor, to solve a single problem. This may be especially appropriate during non-peak hours when many machines are idle and large jobs, such as number crunching, need to be run.

The premise for the use of a network of computers as a loosely coupled multiprocessor is that important, expensive problems can make effective use of this parallel computing environment. This appears to be the case. In particular, it appears that many important problems in numerical computation can be effectively solved by coarse grain parallel algorithms that predominately involve independent concurrent processing and require only a small amount of interprocess communication, shared data, and process creation and termination. Examples of such problems include problems from optimization, VLSI design, and differential equations (see e.g. Feijoo and Meyer [1984], McBryan and Van de Velde [1985], Schnabel [1985], Seitz [1985]). Thus, many problems appear well suited for concurrent implementation in a loosely coupled parallel computing environment, such as a network of computers, where interprocess communication is considerably slower than each processor's computation speed.

In order to effectively use a network of computers as a loosely coupled multiprocessor, it is necessary to have support for interprocess communication, and software that makes distributed, concurrent programs easy to write. Several major projects, all initiated before the release of Berkeley Unix 4.2, have provided this sort of support. They include the Crystal project at the University of Wisconsin (Cook et al [1983a, b]), the Locus project at UCLA (Popek et al [1981], Walker et al [1983]), the Eden project at the University of Washington (Lazowska et al [1981], Almes et al [1985]), the Spice project at Carnegie-Mellon University, and several projects at Xerox PARC (Birrell et al [1981], Shoch and Hupp [1982]). In contrast, DPUP is a far simpler system that builds upon the interprocess communication primitives in Berkeley Unix 4.2.

The 4.2 release of the Berkeley Unix operating system was the first major operating system to provide networking support for commercially available hardware devices. It includes a library of low level interprocess communication primitives and a kernel implmentation of the TCP/IP protocol. The fundamental abstracton provided is the socket, a generalization of the Unix pipe. There are two types of sockets: the stream socket, a reliable communication medium between two processes (on the same or different machines) based upon the TCP protocol, and the datagram socket, which can broadcast to an arbitrary number of processes, using the UDP protocol. The stream socket provides a flow controlled byte stream which is guaranteed to be reliable, in that data will be delivered without error or duplication in the order that it is sent. The datagram socket is not guaranteed to be reliable although practice has shown that usually it is.

To write distributed concurrent programs, it is desirable to have higher level operations than the ones provided in Berkeley Unix. Obvious desirable features include the ability to easily create remote processes and establish communication paths between them, the ability to signal or kill remote processes, and the ability to send and receive data between processes easily. More sophisticated features would include update protocols on data that is sent between processes, and features that facilitate the debugging and testing of distributed concurrent programs.

The DPUP system provides such facilities at a fairly basic level. In conjunction, it provides two models of distributed computation. These are a master/slave model based upon stream sockets, and a broadcast model based upon datagram sockets. In both models, basic operations for initiating and terminating remote processes, and sending and receiving data between these processes, are supported. Some simple update protocols for communicated data are provided. Only very rudimentary debugging support (beyond the standard Unix support such as DBX) is included.

An advantage of the simple approach taken in DPUP is that the system is portable to any Berkeley Unix 4.2 (or equivalent) environment. At the University of Colorado, DPUP has been used on networks of Sun workstations (Sun-2's or 3's), on Vaxes, on Pyramids, and between various combinations of these machines. The concurrent distributed algorithms that have been implemented using DPUP are in areas including global optimization (Byrd et al [1986]), discrete optimization (Trienekens [1986]), VLSI design (Moceyunas [1986]), and solving systems of equations. The DPUP system also has been ported to many other machines including Apollo, Celerity, Gould, ISI, Masscomp, MicroVax, and Symmetric, and the Sequent multiprocessor. A second, more sophisticated system built at the University of Colorado, which is not portable because it includes kernel modifications to support shared memory, is described in Harter and Maybee [1985]. Other systems that support the use of a network of computers for distributed concurrent computation include Carriero and Gelernter [1986], Cooper [1982], Su et al [1985], and Theimer et al [1985].

The remainder of this paper describes the facilities provided in DPUP and gives examples of their use. Section 2 describes the two models of computation, point to point and broadcast, supported in DPUP. In Sections 3 and 4 we discuss the utilities DPUP provides along with each of these models of computation. Section 5 briefly summarizes experience at the University of Colorado using DPUP to implement and test distributed concurrent applications. A simple example of a distributed concurrent computation coded in both the point to point and broadcast modes is given in Appendix A.

Additional information about the DPUP system, including detailed descriptions of the DPUP functions, sample programs, and the DPUP source code, is available from the authors.

2. Models of Computation

The DPUP System is primarily intended to support two models of distributed concurrent computation. The first is a master/slave model and uses point to point communications based upon stream sockets. The second is a broadcast model based upon the datagram sockets. This section briefly describes these two models, their advantages and disadvantages, and the systems architectures underlying them. While it is possible to construct other distributed computation environments using DPUP, for example by combining the point to point and broadcast facilities, we do not discuss such possibilities in any detail.

The master/slave model is a simple model of concurrent computation. The computation is organized around one master process which creates an almost arbitrary number of slaves. Each slave is connected (via a stream socket) to the master, and any communication between slaves is done through the master. Actually, it is possible, using DPUP, for any slave process itself to act as a submaster and create its slaves, so that an arbitrary tree structure is possible. The distributed applications that have used DPUP have not used this generality and it is not discussed further here.

A high level diagram of the architecture underlying the DPUP master/slave model is given in Figure 2.1.

Each machine that is a part of the distributed computation has a server process, called *dp_server*. Slave processes are created by the master as children of the *dp_server* process on the remote host. Point to point communication paths then are created directly between the master process and the slave, so that subsequent communication bypasses the *dp_server*.

The master/slave model is appropriate for many distributed concurrent computations. It is simple to understand and use, and for many applications the centralized control mirrors the natural structure of the parallel algorithm. All the distributed applications projects at the University of Colorado mentioned in Section 1, in optimization, VLSI design, and other areas, have used this model.

The master/slave model has several disadvantages, however. The main disadvantage is that in parallel algorithms where direct communication between slaves (as opposed to communication between a slave and the controlling process) is involved, requiring all communication to go through the master is unnecessary and may create a bottleneck. In particular, in applications where neighboring processes need to communicate, or in applications where each process needs to send data to all processes, the master/slave model may be inefficient. Another disadvantage is that various operating system constraints usually limit the number of point to point connections (file descriptors) per process, so that the master/slave model doesn't scale up indefinitely. Finally, using the master/slave model usually results in a certain degree of synchronization between the slaves and the master, which inevitably causes the slaves to be idle at times in an environment where interprocessor communication is slow. For some parallel algorithms, such as the chaotic relaxation method for solving systems of linear equations which is used as our example in Appendix A, this synchronization is unnecessary because it is not critical that all processes have up-to-date values of all distributed shared variables at all times. Thus using the master/slave model may introduce needless inefficiencies into distributed implementation of such algorithms.

P_i	Processor i
DP	Dp_server
BC	Bc_server
M	Master Process
S_i	Slave Processes
I	Initiator Process
C_i	Colleaque Processes
BCM	Broadcast Medium (Datagram Socket Port)
___	Stream Socket
.....	Parent/Child Fork
----	Datagram Socket

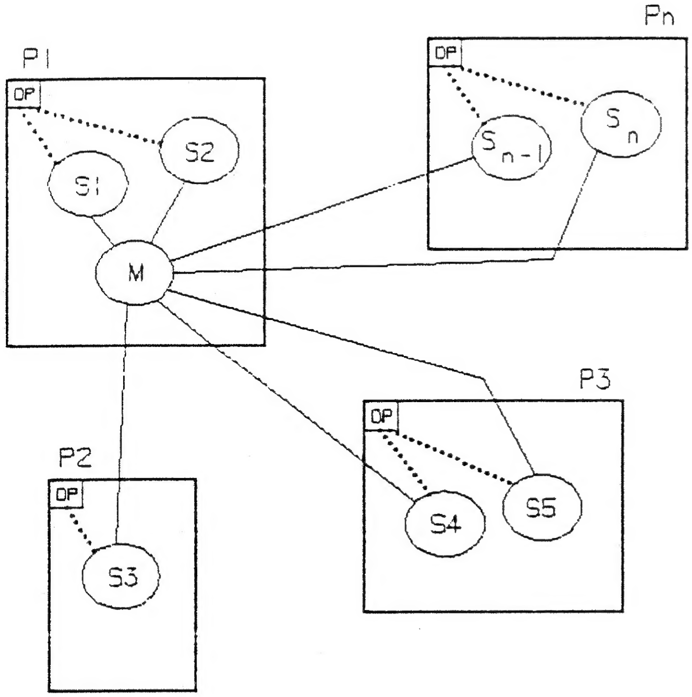

Figure 2.1 Point-to-Point Model

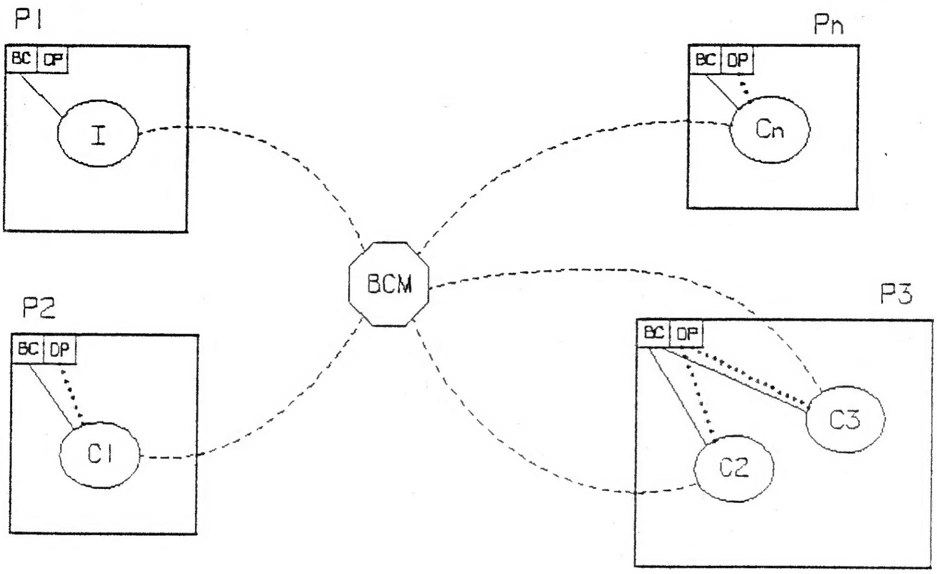

Figure 2.2 Broadcast Model

The broadcast model supported by DPUP provides an alternative to the master/slave model that is especially suited to loosely coupled asynchronous algorithms. The model assumes that all processes are equal, with no master process. (It often is convenient to have a control process to initiate the distributed algorithm, monitor its progress, and decide when to terminate it. However this control process doesn't relay messages between the other processes as is the case in the master/slave.) In the simplest use of the broadcast model, all processes can be thought to be connected on one common communications path, and whenever one process sends a message, all the other processes hear and receive it. Actually, it is possible to have communications between subgroups of processes as well.

A high level diagram of the architecture underlying the broadcast model is given in Figure 2.2. Notice that each computer that is part of the distributed computation now has two servers, the *dp_server* used in the master/slave model and a broadcast server, called *bc_server*. The *dp_server* is used to create processes and help connect them properly to the broadcast system. The broadcast server handles all the broadcast communication for that node. It contains copies of all the variables that are shared through the broadcast system (called "broadcast variables"); the application processes contain their own copies of the broadcast variables as well. Whenever data is broadcast, it is received by the *bc_server* (and not directly by the applications processes). The application processes then query their *bc_server* when they are interested in the values of their broadcast variables. When broadcast data is sent by an application process, it goes directly to the datagram socket (bypassing its own *bc_server*), to be received by all *bc_servers* including its own. There are facilities in the broadcast system for multiple applications to simultaneously use the broadcast system, and for multiple broadcast groups to exist within a single application. These are described in more detail in Section 4.

When using the broadcast system, it is possible that all application processes may not contain up-to-date values of all broadcast variables at any given time. It is also possible that a given update of broadcast variables may never reach one or more *bc_servers* (and hence all application processes on their computers) due either to the unreliable nature of the datagram socket or to the prototypical implementation of the DPUP broadcast system. In particular, the DPUP broadcast system sometimes discards messages if several messages are sent in close succession. In the applications for which the broadcast system is primarily intended, this should not be a problem. The main application of the broadcast system should be for coarse grain parallel algorithms where interprocessor communication is infrequent. In our experience, such algorithms rarely lose broadcast messages. Furthermore, many such applications are iterative and if a message is lost, this does not seriously affect the overall efficiency of the calculation. The iterative calculation proceeds using some out of date values, and eventually these are updated by newer values of the same variables.

3. Point-to-Point Facilities

The point to point facilities in DPUP establish direct communication paths between pairs of processes on distinct (or the same) computers. Generally, they are used to organize a distributed computation in a master/slave hierarchy as discussed in Section 2. In addition, slave processes may act as submasters and spawn their own slaves, so that an arbitrary tree structure is possible. In fact, it is possible to establish communication paths between any pair of processes, thereby enabling the implementation of any communications network.

The DPUP point to point functions are:

dp_create_proc	create a process
dp_rmt_setup	start up a remote process
dp_read	read data
dp_write	send data
dp_kill_proc	kill a process
dp_sig_proc	signal a process
dp_status	display status information
dp_close_proc	close a socket in the master process
dp_rmt_exit	exit a remote process

This section outlines the capabilities and use of these functions. More detailed information is contained in the Unix manual pages for the DPUP functions which are available from the authors.

3.1 Using The System

Each computer in the distributed computation must contain a server process, called *dp_server*. This server is used by the master process to create processes on remote machines, and to establish communications directly between the master and slaves.

Before a user program can create a process on a remote host, *dp_server* must be started on the remote host. The *dp_server* uses a hardwired service port number, usually assigned by the system administrator and placed in the */etc/services* file to support the entire DPUP library. An identical DPUP service port entry should exist in the */etc/services* file on every machine that intends to run the *dp_server*. This server should be started in the background on each of the machines to be used by a distributed concurrent program. This is usually done by adding a line to the file */etc/rc.local* that is executed at boot time. *dp_server*, when running, listens for service requests from user programs on any of the participating machines.

In order to access the DPUP data structures, the applications program must include the file *dpup_user.h* which usually resides in the */usr/local/include* directory. In addition, the application program needs to be linked with the library *libdpup_funs.a*. This usually resides in */usr/local/lib* and is accessed by adding the flag *-ldpup_funs* to the compile command line.

3.2 Remote Process Creation

Remote processes are created using the function call:

```
proc_fd = dp_create_proc(r_stat, host, proc, args, 0);
```

where the process *proc* with arguments *args* is started on *host*. *r_stat* is a pointer to a data structure that is filled by the create function with status information about the process created. The *dp_create_proc* function call returns the file descriptor of a bidirectional pipe for communication with the remote process. In the case of an error, the value −1 is returned. An error will occur if the *dp_server* process is not running on the remote host or if the file *proc* does not exist or cannot be executed on the remote host.

The above example is the simplest form of the *dp_create_proc* function. Actually the arguments *host, proc, args* each can be arrays of equal size. In this case the arrays contain pointers to a list of possible hosts, the corresponding process to start on each host, and the arguments to pass to each process. The call then determines the least utilized machine among the entries in the *host* array, utilizing the Unix load average information, and creates the corresponding process from the array *proc* there. This primitive form of load balancing can be used, if the number of processes is much larger than the number of processors, to attempt an equitable scheduling of resources.

3.3 Remote Process Startup

Using the *dp_create_proc* function causes a remote process to be executed, but does not connect that process to the calling program. To complete this connection the *dp_rmt_setup* function is used. *dp_rmt_setup* should be the first statement executed by the remote process, as follows:

```
create_fd = dp_rmt_setup (&argc, argv)
```

The call returns a file descriptor of a bidirectional pipe that connects to the creator process. The arguments *argc*, and *argv* are the same arguments provided to a main C program. Thus the remote process must define *argc* and *argv* even if it does not intend to use them, because *dp_create_proc* uses them to pass along the process identification handle.

3.4 Data Transfer

Data transfer can be accomplished in a number of ways. In addition to the Berkeley Unix system calls:

```
write(fd, buf, data_size);
read(fd, buf, data_size);
send(fd, buf, data_size, flags);
recv(fd, buf, data_size, flags);
```

the DPUP functions *dp_write* and *dp_read* may also be used:

```
dp_write(fd, buf, data_size);
dp_read(fd, buf, data_size, block_flag);
```

The advantages of the DPUP functions over the system functions are that *dp_read* allows the user to specify that the read should block until *data_size* bytes of data have been returned or to specify that *dp_read* should return immediately with only the data currently available.

Due to the way the DPUP data transfer functions are implemented, it is not possible to send data with a system function and receive the data with a DPUP function, for example, it is not possible to use *dp_read* to receive data sent using *send.*

An additional method of data transfer is by way of the broadcast facilities. These are discussed in Section 4.

3.5 Remote Process Signaling

A master process may send signals to remote processes using the function call:

```
dp_sig_proc(r_stat, sig);
```

This function sends the signal *sig* to the remote process identified by *r_stat.* The function call:

```
dp_kill_proc(r_stat);
```

is a special form of *dp_sig_proc* that sends the KILL signal to the remote process identified by *r_stat.* This function provides the master process a convenient method of terminating remote processes and should be used to clean up after a distributed program is finished to prevent "zombie" processes.

3.6 Remote Exit

At the end of execution the remote process should exit using:

```
        dp_rmt_exit()
```

This function will make a clean exit of the remote process, ie. sockets and file descriptors are closed properly.

3.7 Close Master Sockets

It sometimes is necessary for the master process to close sockets to slaves that have exited, in order not to exceed the system imposeed limit on file descriptors.

To accomplish this the master calls:

```
        dp_close_sock(proc_fd)
```

This should only be done when communication with the remote process is complete.

3.8 Status Information

One of the primary issues in developing distributed programs is trying to diagnose problems occurring between remote processes. When used in the standard manner, the debuggers in Berkeley Unix cannot debug child processes and cannot see communication packets between communicating processes. For this reason, a DPUP function call:

```
        dp_status();
```

has been included.

dp_status is used to provide information relating to communications between distributed processes using the DPUP functions. This call is mainly intended to provide status information, such as a systems call error or a DPUP function error.

3.9 Input/Output Multiplexing

An important facility in developing distributed applications is the ability to multiplex I/O requests among multiple remote processes. In a master/slave application, the master can use the Unix *select* system call to determine which of its slaves want to read data, to write data or have exceptional conditions pending (EOF for example) by using the call:

select(nfds, &readfds, &writefds, &execptfds, &timeout);

nfds is the range of file descriptors, that is, at least one greater (because C counts from 0) than the highest value a file descriptor can assume.

readfds, writefds, exceptfds are bit masks that indicate which client processes (file descriptors) are ready to read/write/take exception. File descriptor 0 corresponds to a 1 in the least significant or 0th bit, file descriptor 1 to the first bit etc. The masks are usually formed by a bitwise OR operation, for example if the server had file descriptors 3, 5, 6, 8 open to client processes, the mask to inquire about these processes would be (as a binary number):

...00101101000

These masks are usually formed in an applications program by an instruction of the form :

mask = mask ¦ (1 << fd);

A timeout value may be specified if the select is not to wait indefinitely for input/output requests. If *timeout* is set to 0, the select takes the form of a *poll*, returning immediately; if the *timeout* parameter is a null pointer, the selection will block indefinitely[1]. *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of −1 is returned and the system parameter *errno* is set to EINTR.

[1] To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

The *select* function cannot be used for I/O multiplexing with the DPUP broadcast facilities.

4. Broadcast Facilities

The broadcast facilities in DPUP provide an alternative to the point-to-point facilities of the previous section. As discussed in Section 2, the point-to-point facilities do not scale up to an arbitrary number of processes due to system constraints. In addition, the master/slave model that the point to point facilities generally are used to implement may not be appropriate for some applications, where it causes a bottleneck at the master or leads to unnecessary synchronization. To alleviate these problems and provide an environment especially appropriate for asynchronous concurrent computations, the DPUP broadcast functions were developed.

The broadcast functions provide an interface to a datagram based broadcast system. Groups of communicating processes sharing the same data and using the broadcast facilities are identified as broadcast groups. The broadcast system supports multiple broadcast groups at one time; processes may be members of more than one broadcast group, or more commonly, several distributed applications that use the broadcast facilities can run on the same system at the same time, each members of a separate broadcast group.

The DPUP broadcast functions are:

bc_open	open a connection to the broadcast server
bc_close	close the connection to the server
bc_create_grp	create a new broadcast group
bc_remove_grp	remove a group
bc_join_grp	join a group
bc_resign_grp	resign from a group
bc_send	broadcast data
bc_receive	receive data

A brief explanation of the function and use of each is given below. For a more detailed specification of a function's arguments, return value, or error diagnostics, consult the specific

DPUP manual entry.

4.1 Using the System

Each computer using the broadcast system must contain two server processes, the *dp_server* discussed in the previous section and a broadcast server, *bc_server.* The *dp_server* is used in creating processes and connecting them to the broadcast system, as shown in Figure 2.2. The broadcast server maintains variables which function as segments of distributed shared memory. Each segment is used by a different broadcast group to communicate with its members. As a segment is created, the number of variables and the length of each variable are specified. Data is broadcast by sending new values to a datagram socket which is read by the *bc_server* processes on each machine; applications processes then read the data from their local *bc_server* Updating a block of variables is not an atomic operation, (see Sections 4.9 and 4.11) so that update protocols may be required. The current update strategy is to overwrite all variables transmitted; a user may choose to implement other strategies. The applications program need not transmit or receive all variables in a broadcast group, the broadcast variable flags allow this to be controlled by the user.

Before a user program utilizes the broadcast system, the *dp_server* and *bc_server* must be running on each participating machine. This is usually done by adding a line to the file */etc/rc.local* that is executed at boot time. A service port must also exist in */etc/services* for both servers. This is described in more detail in Section 3.1.

In order to access the DPUP data structures, the applications programs must include the file *dpup_user.h* which in most installations would reside in the directory */usr/local/include.* In addition, when an applications program is linked, the libraries *libdp_funs.a* and *libbc_funs.a* must be included. These would usually reside in the directory */usr/local/lib* and be accessed by adding the flags *-ldp_funs* and *-lbc_funs* to the compile command line.

4.2 Opening the Broadcast System

Before a process can make use of the DPUP broadcast system, the process must establish a connection to the local broadcast server. This is accomplished by using the following function call:

```
status = bc_open();
```

The return value *status* is 0 if the call was successful; -1 otherwise. This results in a datagram socket for broadcasting data and a stream socket for reading data from the server.

4.3 Closing the Broadcast System

Before an application program that has opened the broadcast system terminates, it should close the system. This closes open socket connections and is important since the total number of such connections to the *bc_server* is limited by the operating system. This is accomplished by using the function call:

```
status = bc_close();
```

function. The return value *status* is 0 if the call was successful; -1 otherwise.

4.4 Creating a Broadcast Group

To create a new broadcast group in the DPUP broadcast system, the following function call is used:

```
status = bc_create_grp(bc_id, var_table, num_vars);
```

This function requests the local broadcast server to set up a new broadcast group containing *num_vars* broadcast variables whose sizes are specified in *var_table.* A group identifier will be assigned by the local broadcast server and returned in the *bc_id* argument. The return value *status* is 0 if the call was successful; -1 otherwise. The local broadcast server will broadcast the new group's parameters and id thus informing the *bc_server*s on other hosts of the existence of the new group. The other servers then create their own copies of the broadcast variables.

4.5 Remote Process Creation

After an initiating process has created the broadcast group, remote processes may be created. This is done using the point-to-point function call *dp_create_proc* described in Section 3.2. The initiating process must also send the group id number to the remote process using *dp_write.* The remote process must begin with a *dp_rmt_setup*, as described in Section 3.3, and then read the group id with a *dp_read.* At the end of its execution the remote process should exit using *dp_rmt_exit.*

4.6 Joining a Broadcast Group

Once a broadcast group has been created, processes that have opened the broadcast system may join the group by using the function call:

```
id_index = bc_join_grp(bc_id);
```

(The process that created the group also must join it). This function informs the local broadcast server that a new process has joined the broadcast group identified by *bc_id.* The function returns a broadcast identifier index that identifies the broadcast group. The process then may transmit and receive data within this group. Processes may be members of multiple broadcast groups at the same time.

4.7 Resigning from a Broadcast Group

A process may resign from a broadcast group identified by id_index by using the function call:

```
result = bc_resign_grp(id_index);
```

Once a process resigns from a broadcast group, the process may no longer transmit or receive broadcast data within that group unless the process once again joins the group.

If a process is the last member of a group to resign, and if another process has a *bc_remove_grp* command pending, the group will be removed at this time.

4.8 Removing a Broadcast Group

To remove a broadcast group from the DPUP broadcast system, the following function is used:

```
result = bc_remove_grp(bc_id);
```

bc_id identifies the broadcast group that is to be removed. Currently any process may call this function (i.e. there is no concept of superuser processes in the broadcast system at this time). It should be noted that removing a broadcast group while the group is still active will be delayed until all the members of the group have resigned and the group is no longer active. It is important that non-active groups be removed so that the broadcast server state tables do not become full of non-active groups.

4.9 Broadcasting Data

A process sends broadcast data using the function:

```
bytessent = bc_send(id_index, var_flags, data);
```

This function broadcasts variables within the broadcast group specified by *id_index*. *var_flags* is an array used to specify which variables in this group are being transmitted. A variable will be transmitted only if the associated *var_flag* is set to 1. All broadcast group data variables must first be moved, if necessary, to a contiguous block of memory specified by *data*. The return value *bytessent* is the number of bytes sent if the call was successful; -1 otherwise.

This is not an atomic operation, that is, the data is sent in three parts, the header, the varflags, and the actual variables. Therefore, the data sent by one process may be interleaved with data from other processes. In the current implementation, once the header packet is received, the server awaits the two remaining parts and discards any data received from other processes until the transaction is completed or a time limit is reached. Thus, it is possible that some broadcast information will never be received; this is more likely the more frequently broadcast data is sent. It is not possible, in the current implementation, to notify other sending processes that their data may have been discarded.

4.10 Receiving Broadcast Data

A process receives broadcast data using the function:

```
bytesrcvd = bc_receive(id_index, mode, data, var_flags);
```

This function is used to read broadcast data variables associated with the broadcast group identified by *id_index* that have been received by the local broadcast server. The function argument *mode* indicates which variables are to be read. The following modes are currently implemented:

mode	description
0	all variables are returned
1	only new variables are returned
2	all variables requested by *var_flags* are returned
3	only requested variables that are new are returned
4	block until requested variables arrive

New variables are defined as variables that the process has not yet read. Upon return, the argument *var_flags* is updated to indicate which variables have been read and the data is placed in the location *data*. The return value *bytesrcvd* is the number of bytes received if the call was successful; -1 otherwise. Mode 4 does not include a timeout parameter, and therefore if the processes are not synchronized properly, this mode can block forever.

4.11 Timing Considerations

The broadcast system is sensitive to timing. As discussed in Section 4, it was created primarily for applications where interprocessor communication is infrequent. Descriptions of the timing problems and their current solutions follow. The next generation of distributed utilities is intended to alleviate these problems.

The interactions of the group commands, creation, joining, resigning and removal, can cause one process to try to join a non-existent group or to try to remove an active group. The *bc_join_grp* function has an error return that must be checked by the user program to see if the join was successful; if not, it should be tried again. The *bc_remove_grp* function blocks until the group is inactive. These have proved to be adequate solutions to these problems.

There can be problems synchronizing the creation and initialization of data in a broadcast group with requests to read the data. Typically one process creates and initializes the group. If other processes don't wait until this step is complete,they may request data and receive uninitialized values. This can be avoided by reading data in mode 4 the first time, so the process blocks until there is actually data available. This has proven an adequate solution to this problem.

In order to deal with the problems of non-atomic data mentioned in Section 4.8, the *bc_server* may take somewhat drastic action. As explained in Section 4.8, data broadcast by members of the group may or may not be received and recorded by one or more *bc_servers*. This action is a local one, taken by each *bc_server*, yet the *bc_send* is a global operation. The result is that different computers participating in a distributed computation may have different values of the broadcast variables, even if no messages remain to be read. Therefore, the current implementation of broadcast variables is best suited to the type of coarse grain parallel algorithms discussed at the end of Section 2.

5. Experience Using DPUP

Several parallel computation projects at the University of Colorado have successfully implemented and tested distributed concurrent applications algorithms built upon DPUP. Almost all these algorithms use the point to point facilities described in Section 3, because they were available to users earlier than the broadcast facilities. This section briefly summarizes some of this research.

The predominant use of DPUP has been in the development of parallel algorithms for problems from optimization and VLSI design. Many problems in these fields seem to be amenable to solution by coarse grain parallel algorithms that require little shared data or interprocess communication. Thus they appear to be good candidates for efficient parallel implementation on a network of computers.

Byrd et al [1986] have used DPUP to construct several concurrent global optimization algorithms. The global optimization problem is to find the lowest minimum of a function of real variables that may have multiple local minima, i.e. lowest points in some region of the variable space. This problem is difficult and expensive to solve and thus parallel algorithms are of interest. Byrd et al propose a synchronous stochastic method that has three main parts: a Monte Carlo search of the variable space, a phase that essentially performs nearest neighbor calculations, and a phase where multiple local minimizations are conducted concurrently. The parallelism is at a very high level and relatively few messages between processes are required. Several methods of this type have been implemented using DPUP and tested on networks of 4 and 8 Sun-3 workstations, with encouraging results. Speedups are often 80% of optimal or higher on problems where function evaluation is expensive.

Trienekens [1986] used DPUP to solve two discrete optimization problems, the knapsack problem and the traveling salesman problem, on a network of computers. These problems frequently are solved by branch and bound methods which dynamically construct a tree of simpler subproblems, from whose solutions the solution of the original problem is obtained. The parallel methods use the master process to generate the tree of subproblems and monitor progress, and distribute the solution of the subproblems to the various computers on the network. On a 75 city traveling salesman problem, an implementation of the parallel algorithm using DPUP ran over 4.6 times as fast on 5 Pyramid P90-X computers as the same algorithm on one Pyramid.

Several projects in parallel algorithms for VLSI design problems have been based upon DPUP. Moceyunas [1986] has used DPUP to construct parallel simulated annealing algorithms for the optimal placement of a VLSI chip. This work has shown that simulated annealing can, through several partitioning strategies, gain performance advantages from parallel solution on a local area network of computers. Work on parallel algorithms for Boolean function minimization currently is underway. This work is aided by a system built on top of DPUP by Mueller [1986] that allows the network to simulate various interconnection topologies.

DPUP has been used to implement several simple iterative methods for solving systems of linear equations. These include the chaotic relaxation algorithm of Chazan and Miranker [1969] (given as the example in Appendix A in two versions, one using the point to point facilities and the other using the broadcast facilities), and block Gauss-Seidel and SOR algorithms for solving systems arising from elliptic partial differential equations.

DPUP also was used to build a tool for evaluating the speed and completeness of various vendors' Ethernet hardware and Berkeley Unix interprocess communication implementations. This benchmarking tool was run in both loopback mode and remote mode on all machines claiming to have Berkeley Unix interprocess communication at the Portland Usenix conference vendor exhibit in June 1985. It discovered incomplete and incorrect implementations in several vendor products. The machines tested were Apollo, Celerity, Gould, ISI, Masscomp, MicroVax, Pyramid, Sequent, Sun, Symmetric, and Vax. This use demonstrates the portability of DPUP.

6. References

G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe [1985], "The Eden system : a technical review," *IEEE Transactions on Software Engineering* 11, pp. 43-59.

A. Birrell, R. Levin, R. Needham, M. Schroeder [1981], "Grapevine: an exercise in distributed computing," *Proc. Eighth Symposium on Operating Systems Principles,* pp. 169-177.

R. H. Byrd, C. L. Dert, A. H. G. Rinnooy Kan, and R. B. Schnabel [1986], "Concurrent stochastic methods for global optimization", Technical Report CU-CS-338-86, Department of Computer Science, University of Colorado at Boulder.

N. Carriero and D. Gelernter [1986], "The S/Net's Linda kernel," *ACM Transactions on Computer Systems* 4, pp. 110-129.

D. Chazan and W. Miranker [1969], "Chaotic relaxation," *Linear Algebra and its Applications,* 2, pp. 199-222.

R. Cook, R. Finkel, D. DeWitt, L. Landweber, R. Virgilio [1983a], "The Crystal nugget, Part I of the first report of the Crystal project," Technical Report 499, Computer Sciences Department, University of Wisconsin - Madison.

R. Cook, R. Finkel, D. DeWitt, L. Landweber, R. Virgilio [1983b], "The Crystal nugget, Part II of the first report of the Crystal project," Technical Report 500, Computer Sciences Department, University of Wisconsin - Madison.

E. C. Cooper [1982], "Writing distributed programs with Courier," Technical Report, Computer Science Division, University of California at Berkeley.

B. Feijoo and R. R. Meyer [1984], "Piecewise-linear approximation methods for nonseparable convex optimization", Technical Report No. 521, Computer Sciences Department, University of Wisconsin - Madison.

P. K. Harter and P. Maybee [1985], "DCS : A system for distributed computing support," Department of Computer Science Technical Report CU-CS-309-85, University of Colorado at Boulder.

E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, S. Vestal [1981], "The architecture of the Eden system," *Proc. Eighth Symposium on Operating Systems Principles,* pp. 148-159.

O. McBryan and E. Van de Velde [1985], "Elliptic and hyperbolic equation solution on hypercube multiprocessors, Courant Institute Preprint, Aug. 1985.

P. Moceyunas [1986], M.S. Thesis, Department of Electrical Engineering, University of Colorado at Boulder, in preparation.

H. Mueller [1986], "Multi-processor emulation on a local area network," M.S. Thesis, Department of Computer Science, University of Colorado at Boulder.

G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudish, G. Theil [1981], "LOCUS: A network transparent, high reliability distributed system," *Proc. Eighth Symposium on Operating Systems Principles,* pp. 169-177.

R. B. Schnabel [1984], "Parallel computing in optimization," in *Computational Mathematical Programming,* K. Schittkowski, ed., Springer-Verlag, Berlin, pp. 357-382.

C. L. Seitz [1985], "The Cosmic Cube," *Communications of the ACM* 28, pp. 22-33.

J. Shoch and J. Hupp [1982], "The "worm" programs - early experience with a distributed computation," *Communications of the ACM 25,* pp. 172-180.

W.-K. Su, R. Faucette, and C. Seitz [1985], "C programmers guide to the cosmic cube," Department of Computer Science Technical Report 5203:TR:85, California Institute of Technology.

M. M. Theimer, K. A. Lantz, and D. R. Cheriton [1985], "Preemptable remote execution facilities for the V-system," *Proc. Tenth ACM Symposium on Operating Systems Principles,* Dec. 1985, pp. 2-12.

H. W. J. M. Trienekens [1986], "Parallel branch and bound methods for discrete optimization problems," (tentative title), in preparation.

B. Walker, G. Popek, R. English, C. Kline, and G. Thiel [1983], "The LOCUS distributed operating system," *Proc. of the Ninth ACM Symposium on Operating Systems Principles,* Oct. 1983, pp. 49-70.

Appendix A. Examples using Point-to-Point and Broadcast Facilities

This appendix contains two implementations, using DPUP, of a simple but nontrivial distributed concurrent computation. The first implementation uses the point-to-point facilities to build a master/slave version of the algorithm. The second implementation is a broadcast version of the same algorithm.

The algorithm implemented is a "chaotic relaxation" method for solving systems of n linear equations in n unknowns $Ax=b$. This method, first proposed by Chazan and Miranker[1969], is an asynchronous version of the Gauss-Seidel method. The Gauss-Seidel method simply cycles through the n equations in order, solving each equation to yield a new value for the corresponding variable. That is, when solving the k^{th} equation, the current values of x[i], $i \neq k$, are substituted into this equation and a new value of x[k] is obtained. For certain classes of matrices A, the iterates converge to the solution of the linear system.

In the chaotic version of the Gauss-Seidel method, the order of processing the equations is arbitrary, and the computation for processing the k^{th} equation may use arbitrarily old values of the other variables. Chazan and Miranker have shown necessary conditions for such an algorithm to converge to the correct solution.

A simple parallel version of the chaotic relaxation algorithm is obtained by creating one process to handle each equation. Process k repeatedly solves the k^{th} equation for the k^{th} variable, using whatever values of the other variables it currently has, and then sends its new value of x[k] to the other processes. This is the method implemented below. In the point-to-point version, each process repeatedly sends its new value of its variable to the master, obtains the master's latest values of the other variables. and performs its next iteration. The master monitors the computation and decides when to terminate it. In the broadcast version, each process repeatedly broadcasts its new value of its variable, obtains the values of the other variables from its *bc_server*, and performs its next iteration. An initiating process is used to start, monitor, and terminate the algorithm.

Each version of the example contains five procedures. *Main* is the driver which includes the creation of the remote processes and the communication structures. *Proc_ctrler* monitors the computation and, in the point-to-point example, is the master in the master/slave communications pattern. *Chaos_rmt* is the remote process that repeatedly solves the k^{th} equation for the k^{th} variable. The input and output procedures, *input_data* and *output_rlts* are included for completeness; they do not contain any DPUP function calls.

```
/*
**      POINT-TO-POINT VERSION OF THE CHAOTIC RELAXATION ALGORITHM
**
**      chaos_0.h : include file
**
**      Description :
**                  This file contains the parameters used by the master and
**      the remote processes . In particular :
**      - the names of the output file, the remote process;
**      - the number of different hosts on which the remote processes will
**      be started;
**      - the constants related to the number of equations in the system;
**      - the value used for the stopping condition;
**
**      Timothy Gardner - March 1984
**      Carla Mowers and Isabelle Gerard - June 1986
*/

#define OUTPUT_FILE       "/tools/dpup/examples/strm/chaos0_output"

/* name of remote processes */
#define  PROC_NAME        "/tools/dpup/examples/strm/chaos_rmt_0"

/*
** number of different hosts on which you want to run the remote processes
** ( maximum 7 )
*/
#define NUMHOSTS 7

/* name of the hosts */
#define  HOST_NAME0     "molson"
#define  HOST_NAME1     "bass"
#define  HOST_NAME2     "watneys"
#define  HOST_NAME3     "heineken"
#define  HOST_NAME4     "anchor"
#define  HOST_NAME5     "guiness"
#define  HOST_NAME6     "becks"

/* maximum number of processes which can be started */
#define  MAXNPROC        32

/*
** a solution has been found when all old x values differ from the
** new x values by no more than          RLT_PRECISION
*/

#define   RLT_PRECISION  0.0001

/*
**      POINT-TO-POINT VERSION OF THE CHAOTIC RELAXATION ALGORITHM
**
**      This is the master process
**
**      Usage: chaos data_file
**
**      Description :
**                  The purpose of this program is to find the result X of the
**      equation AX = B, where A is an n*n matrix, and X and B are vectors
**      of size n.
**                  The master process reads in the values of n, A, and B from
**      the input file. Then it creates n remote processes each of which
**      is going to solve an equation of the form :
**      A[proc_num][] X[proc_num] = B[proc_num] ( where proc_num is the
**      number of the current process ) .
**
```

```
**          Data_file format:
**
**                    n
**                    A11  A12  A13  A14  ....  A1n  B1
**                    A21  A22  A23  A24  ....  A2n  B2
**                     .    .    .    .   ....   .    .
**                     .    .    .    .   ....   .    .
**                     .    .    .    .   ....   .    .
**                    An1  An2  An3  An4  ....  Ann  Bn
**
**          Output:
**
**                    The x values for the solution of the equation: Ax = b
**
**
**          Timothy Gardner - April 1984
**          Carla Mowers and Isabelle Gerard - June 1986
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include "/tools/dpup/src/dpup_user.h"
#include "/tools/dpup/examples/strm/chaos_0.h"

struct proc
{
          int       proc_sock;                        /* remote process socket descriptor */
          int       equ_num;                          /* equation number process is       */
          struct    rmt_stat            host;       /* to receive the name of the host  */
};


char      *Host_nms[2] = {"", NULL};                  /* used to receive the name of the    */
                                                      /* host on which a process is created */

char      *Proc_nm[2] = {PROC_NAME,NULL};             /* used to specify the name of the    */
                                                      /* remote process to be created       */

                                                      /* array of hosts on which remote     */
                                                      /* processes can be started           */
char      *Host_names[8] = {HOST_NAME0, HOST_NAME1, HOST_NAME2,
                            HOST_NAME3, HOST_NAME4, HOST_NAME5,
                            HOST_NAME6, NULL};

int       Steps;                                      /* number of steps to get the result  */

struct    proc      Proc_table[MAXNPROC];             /* to keep information about remote   */
                                                      /* processes created                  */

FILE      *Ofd;                                       /* file descriptor for output file    */

main(argc, argv)                                                                          main
  int     argc;
  char    *argv[];
{
          double    data_array[MAXNPROC][MAXNPROC + 2];            /* array to hold    */
                                                                   /* input data       */
          double    x_array[MAXNPROC];                             /* array to hold    */
```

```
                                                          /* calculated x values*/

        int         num_equ;                    /* number of equations in system       */
        int         procs[MAXNPROC];            /* process numbers indexed by socket   */
                                                /* number
        register int        i;
        int         result;

        Steps = 0;

        /* open the input file */
        if (( Ofd = fopen(OUTPUT_FILE,"w")) < 0 )
        {
                printf("cannot open OUTPUT_FILE\n");
                exit();
        }

        /* read data from the input file */
        input_data(argc, argv, &num_equ, data_array);

        /* start with all x values equal to zero */
        for (i = 0; i < num_equ; i++)
                x_array[i] = 0;

        /*
        **   create one remote process for each equation and send the A and b
        **   values and the initial x values to that process
        */

        for (i = num_equ; i > 0; i--)
        {
                Host_nms[0] = Host_names[(i-1) % NUMHOSTS];
                if ((Proc_table[i - 1].proc_sock =
                        dp_create_proc(&(Proc_table[i - 1].host),
                                Host_nms, Proc_nm, 0))<0)
                {
                        printf("master: create_proc failure\n");
                        dp_status();
                        exit();
                }

                /* send the values of A and B to the remote process */
                result = dp_write(Proc_table[i - 1].proc_sock,
                                (char *) &data_array[i - 1][0], (num_equ+3)*sizeof(double));

                /* send the initial value of X to the remote process */
                result = dp_write(Proc_table[i - 1].proc_sock,
                                (char *) &x_array[0], num_equ*sizeof(double));

                /* record socket descriptor of remote process */
                procs[Proc_table[i - 1].proc_sock] = i - 1;

        }

        /* compute the result X of the equation AX = B */
        proc_ctrler(procs, num_equ, x_array);

        /* output the results */
        output_rlts(num_equ, data_array, x_array);
}
```

```
/**/

/*
**        proc_ctrler – routine to handle the sending and receiving of data
**                        to and from the remote processes
*/
```

proc_ctrler

```
proc_ctrler(procs, num_equ, x_array)
  int               procs[], num_equ;
  double            x_array[];
{
        int       sock;                  /* socket descripter           */
        int       fds_mask;              /* select function masks       */
        int       tmp_mask;              /* saves value of select mask  */
        int       nfds;                  /* number of selected sockets  */
        int       proc_num;              /* process number                  */
        int       done_flags;            /* when all bits are zero a solution    */
                                         /* has been found                         */
                                         /* this variable is updated each time a */
                                         /* remote process returns a new x value */
        double    result;                /* result returned by remote process    */
        double    diff;                  /* diff between new and old x values    */

        register int i;

        struct    timeval timeout;

        timeout.tv_sec  = 0;                      /* used by select for */
        timeout.tv_usec = 0;                      /* polling sockets    */

        /* clear flags – i.e. a cleared flag is a bit set to 1 */
        /* a marked  flag is a bit set to 0 */

        for (i=0; i<num_equ; i++)
                done_flags |= (1 << i);

        while (done_flags > 0)                    /* a solution has not yet been found */
        {
                fds_mask = 0;

                /* set up file descriptor mask used by the select function */
                for (proc_num=0; proc_num < num_equ; proc_num++)
                        fds_mask |= (1 << Proc_table[proc_num].proc_sock);

                /* fds_mask must be restored after each select call */
                /* because select clears fds_mask if no fd is ready */
                /* to be read                  */
                tmp_mask = fds_mask;

                /* wait for remote processes to return results  */
                /* by polling each socket until data is present */

                while ((nfds = select(_NFILE-1, &fds_mask, 0, 0, &timeout))<=0)
                        fds_mask = tmp_mask;

                sock = 0;

                while (nfds)
                {

                        /* determine processes that have returned results */
                        /* by checking each bit in the mask returned by    */
```

...proc_ctrler

```
            /* select.  set bits mark ready to read fd's          */

            for (;;)
            {
                if (fds_mask & 1)
                {
                    fds_mask = fds_mask >> 1;
                    break;
                }
                sock++;
                fds_mask = fds_mask >> 1;
            }

            proc_num = procs[sock];

            /* read result returned from remote process */

            dp_read(Proc_table[proc_num].proc_sock, (char *) &result, sizeof(double), 1);


            diff = result - x_array[proc_num];
            if (diff < 0)
                diff *= -1;

            if (diff <= RLT_PRECISION)

                /* set the done_flag for this process by        */
                /* clearing the proc_num th bit of done_flags   */

                done_flags &= (done_flags ^ (1 << proc_num));
            else
                /* clear the done_flag for this process by      */
                /* setting the proc_num th bit of done_flags    */

                done_flags |= (1 << proc_num);

            /* update the x value array with new x value */

            x_array[proc_num] = result;

            fprintf(Ofd, "step number : %2d          ", Steps);
            fprintf(Ofd, "   x[%d] = %8.8lf\n", proc_num, result);

            fflush(Ofd);

            /* a solution has been found when all process have   */
            /* returned a new x value that is not different by  */
            /* more than RLT_PRECISION

            if (done_flags == 0)
                break;

            /* send new x values to the remote process                  */

            dp_write(Proc_table[proc_num].proc_sock, (char *) x_array, num_equ *sizeof(double));

            Steps++;
            nfds--;
            sock++;
        }
    }
```

```
		/* a solution has been found so terminate remote processes */

		for (i=0; i<num_equ; i++)
			dp_kill_proc(&(Proc_table[i].host));
}

/**/

/*
**		input_data - routine to fetch data from the data file
*/
```

input_data

```
input_data(argc, argv, num_equ, data_array)
  int			argc, *num_equ;
  double		data_array[][MAXNPROC + 2];
  char			*argv[];
{
		register int row, col;
		FILE		*data_fp;

		if (argc < 2)
		{
			printf("usage: %s datafile\n", argv[0]);
			exit();
		}

		if ((data_fp = fopen(argv[1], "r")) == NULL)
		{
			printf("cannot open %s\n", argv[1]);
			exit();
		}

		if (fscanf(data_fp, "%d", num_equ) < 1)
		{
			printf("data format error in %s\n", argv[1]);
			exit();
		}

		for (row=0; row<*num_equ; row++)
		{

			/* place number of variables and the equation number in */
			/* each array along with the A and b values              */

			data_array[row][0] = (double) *num_equ;
			data_array[row][1] = (double) row;

			for (col=2; col<*num_equ + 3; col++)
			{
				if (fscanf(data_fp, "%lf", &data_array[row][col]) < 1)
				{
					printf("data format error in %s\n", argv[1]);
					exit();
				}
			}
		}
}

/**/

/*
**		output_rlts - routine to output the final results
*/
```

output_rlts

```
output_rlts(num_equ, data_array, x_array)
```

```
  int               num_equ;
  double            data_array[][MAXNPROC + 2];
  double            x_array[];
{
        register int i, row, col;

        fprintf(Ofd, "\n\nequation num                    x value\n");
        fprintf(Ofd, "------------                    -------\n");

        for (i=0; i < num_equ; i++)
                fprintf(Ofd, "        %d                      %16.12lf\n", i+1, x_array[i]);

        fflush(Ofd);

        fprintf(Ofd, "\n\n");

        fprintf(Ofd, "program statistics\n");
        fprintf(Ofd, "------- ----------\n\n");

        fflush(Ofd);

        fprintf(Ofd, "number of hosts                              = %d\n", 3);
        fprintf(Ofd, "number of processes                          = %d\n", num_equ);
        fprintf(Ofd, "total number of steps                        = %d\n", Steps);
        fprintf(Ofd, "degree of precision                          = %lf\n",RLT_PRECISION);
        fprintf(Ofd, "\n");

        fflush(Ofd);
}


/*
**      POINT-TO-POINT VERSION OF THE CHAOTIC RELAXATION ALGORITHM
**
**      This is the remote process
**
**      remote process to calculate a new x value for the equation sent
**      from the master process
**
**      Timothy Gardner - April 1984
**      Carla Mowers and Isabelle Gerard - June 1986
*/

#include <stdio.h>
#include "/tools/dpup/examples/strm/chaos_0.h"
```

main

```
main(argc, argv)
  int     argc;
  char    *argv[];
{
        int             sock;           /* socket connection to master      */
        int             length;         /* amount of data read              */
        char            data_buf[500];  /* buffer to store A and b values   */
        char            x_values[500];  /* buffer to store x values         */
        double          sum, num_equ, proc_num;
        double          b_value;
        double          *dbl_ptr, *x_ptr;
        register int    i;

        /* get socket descriptor to master process */
        sock = dp_rmt_setup(&argc, argv);

        if (sock < 0)
        {
                printf("remote: rmt_setup error\n");
                dp_status();
                dp_rmt_exit();
        }

        /* read the values of A, B, and X */
        if (dp_read(sock, data_buf, 0, 1) < 0)
        {
                printf("remote: read error 1\n");
                dp_status();
                dp_rmt_exit();
        }

        dbl_ptr = (double *) data_buf;

        num_equ = dbl_ptr[0];
        proc_num   = dbl_ptr[1];
        b_value    = dbl_ptr[((int) num_equ)+2];

        /* continue processing new x values                          */
        /* when a solution has been found the master will terminate us  */

        for (;;)
        {

                /* read the value of X */

                if (dp_read(sock, (char *) x_values, 0, 1) < 0)
                {
                        printf("remote: read error 2\n");
                        dp_status();
                        dp_rmt_exit();
                }

                x_ptr = (double *) x_values;

                sum = 0;

                for (i = 0; i < num_equ; i++)
                {
                        if ( i != proc_num )
                        {
                                sum += dbl_ptr[i+2] * x_ptr[i];
                        }
                }

                sum = (b_value - sum) / dbl_ptr[((int) proc_num) + 2];

                /* send calculated x value back to the master */

                if (dp_write(sock, (char *) &sum, sizeof(double)) < sizeof(double))
                {
                        printf("remote: dp_write error\n");
                        dp_status();
                        dp_rmt_exit();
                }
        }
}

/*
**      BROADCAST VERSION OF THE CHAOTIC RELAXATION  ALGORITHM
**
**      chaos_1.h : include file
**
**      Description :
**              This file contains the parameters used by the master and
**      the remote processes . In particular :
**      - the names of the output file, the debbuging file, the remote
**      process;
**      - the number of different hosts on which the remote processes will
**      be started;
**      - the constants related to the number of equations in the system;
**      - the value used for the stopping condition;
**      - the structure describing the data of the group;
**
**      Carla Mowers and Isabelle Gerard - June 1986
*/

#define OUTPUT_FILE     "/tools/dpup/examples/dgrm/chaos1_output"

/* name of remote processes */
#define  PROC_NAME      "/tools/dpup/examples/dgrm/chaos_rmt_1"

/*
** number of different hosts on which you want to run the remote processes
** ( maximum 7 )
*/

#define NUMHOSTS 7

/* name of the hosts */
#define  HOST_NAME0     "molson"
#define  HOST_NAME1     "bass"
#define  HOST_NAME2     "guiness"
#define  HOST_NAME3     "anchor"
#define  HOST_NAME4     "heineken"
#define  HOST_NAME5     "becks"
#define  HOST_NAME6     "watneys"

/* maximum number of processes which can be started */
#define  MAXNPROC       32

/* size of the buffers to send and receive data */
#define  BUFSIZE        sizeof(double) * MAXNPROC * MAXNPROC +
                            (2 * sizeof(double) + sizeof(short)) * MAXNPROC

/*
** number of variables in the group */
** MAXNPROC for each line of A, B, X, and done_flag
*/

#define NUMVARS 4 * MAXNPROC
/*
** a solution has been found when all old x values differ from the
** new x values by no more than RLT_PRECISION
*/

#define   RLT_PRECISION  0.0001

/* structure describing the data of the group */

struct chaos
{
        double  A[MAXNPROC][MAXNPROC];  /* matrix A                 */
        double  B[MAXNPROC];            /* vector B                 */
        double  X[MAXNPROC];            /* solution of the equation */
        short   done_flag[MAXNPROC];    /* flag to tell a remote is */
                                        /* done                     */
};
```

```
/*
**        BROADCAST VERSION OF THE CHAOTIC RELAXATION  ALGORITHM
**
**        This is the master process
**
**        Usage: chaos data_file
**
**        Description :
**                  The purpose of this program is to find the result X of the
**        equation AX = B, where A is an n*n matrix, and X and B are vectors
**        of size n.
**                  The master process reads in the values of n, A, B and the
**        initial values of X, from the input file. Then it creates n remote
**        processes each of which is going to solve an equation of the form
**        A[proc_num][] X[proc_num] = B[proc_num] ( where proc_num is the
**        number of the current process ) .
**
**        Input_file format:
**
**                  n
**                  A11  A12  A13  A14  ....  A1n
**                  A21  A22  A23  A24  ....  A2n
**                   .    .    .    .   ....   .
**                   .    .    .    .   ....   .
**                   .    .    .    .   ....   .
**                  An1  An2  An3  An4  ....  Ann
**                  B1   B2   B3   B4   ....  Bn
**                  X1   X2   X3   X4   ....  Xn
**
**        Output:
**
**                  The X values for the solution of the equation: AX = B
**
**
**        Carla Mowers and Isabelle Gerard -- June 1986
*/

#include <stdio.h>
#include <signal.h>
#include "/tools/dpup/src/bdct.h"
#include "/tools/dpup/src/dpup_user.h"
#include "/tools/dpup/examples/dgrm/chaos_1.h"


char      *Host_nms[2] = {"", NULL};          /* to receive the name of the host  */
                                              /* on which a process is created    */

char      *Proc_nm[2] = {PROC_NAME,NULL};     /* used to specify the name of the  */
                                              /* remote process to be created     */

                                              /* array of hosts on which remote */
                                              /* processes can be started       */
char      *Host_names[8] = {HOST_NAME0, HOST_NAME1, HOST_NAME2,
                            HOST_NAME3, HOST_NAME4, HOST_NAME5,
                            HOST_NAME6, NULL};

int       Steps;                              /* number of steps to get the result */

FILE      *Ofd;                               /* file descriptor for output file   */

main(argc,argv)
int       argc;
char      *argv[];
```

main

```
{
        struct chaos *c_ptr;              /* pointer to the structure descri-     */
                                          /* bing the data of the group            */
        int     proc_sock[MAXNPROC];      /* array of the socket to the remote    */
                                          /* processes                              */
        char    data[BUFSIZE];            /* buffer to send/receive data           */
        char    proc_num[MAXNPROC];       /* number of process to be created       */
        short   var_table[NUMVARS];       /* array containing the sizes of the    */
                                          /* data of the group                      */
        char    var_flags[NUMVARS];       /* flags to tell which data to send     */
                                          /* or receive                             */
        int     num_equ;                  /* number of equations                    */
        struct rmt_stat    host;          /* to receive the name of the host on  */
                                          /* which a remote process is created    */
        int     result;                   /* number of bytes sent/received         */
        struct  bd_id id;                 /* id of the broadcast group              */
        short   id_index;                 /* index of the current process in       */
                                          /* the broadcast group                    */
        char    str_num_equ[10];          /* to pass the number of equations to  */
                                          /* the remote processes                   */
        register short i,j;


        c_ptr = (struct chaos *) data;

        Steps = 0;


        /* open the output file */
        if((Ofd = fopen(OUTPUT_FILE,"w")) < 0)
        {
                perror("fopen");
                bc_close();
                exit(1);
        }

        /* initialization of the data of the group */
        input_data(argc,argv, &num_equ, c_ptr);

        /* set the var_flags corresponding to A,B and X, to 1 */
        for (i=0; i < NUMVARS; i++)
        {
                var_flags[i] = 0;
        }

        for ( i = 0; i < *num_equ ; i++ )
        /* each of these flags corresponds to a line of the matrix A */
        {
                var_flags[i] = 1;
        }
        for ( i = MAXNPROC; i < MAXNPROC + *num_equ; i++ )
        /* each of these flags corresponds to an element of the vector B */
        {
                var_flags[i] = 1;
        }
        for ( i = 2 * MAXNPROC; i < 2 * MAXNPROC + *num_equ; i++ )
        /* each of these flags corresponds to an element of the vector X */
        {
                var_flags[i] = 1;
        }

        /* initialize the var_table in which the sizes of the variables */
/* are given                                                           */
for ( i = 0; i < MAXNPROC ; i++ )
/* corresponds to a line of the matrix A */
{
        var_table[i] = sizeof(double) * MAXNPROC;
}
for ( i = MAXNPROC; i < 2 * MAXNPROC; i++ )
/* corresponds to an element of the vector B */
{
        var_table[i] = sizeof(double);
}
for ( i = 2 * MAXNPROC; i < 3 * MAXNPROC; i++ )
/* corresponds to an element of the vector X */
{
        var_table[i] = sizeof(double);
}
for ( i = 3 * MAXNPROC; i < 4 * MAXNPROC; i++ )
/* corresponds to a stop flag for each of the remote processes */
{
        var_table[i] = sizeof(short);
}

/* establish a connection with the local broadcast server */
result = bc_open();
if (result < 0)
{
        printf("bc_open error\n");
        dp_status();
        exit(1);
}

/* set up a new broadcast group */
result = bc_create_grp(&id, var_table, NUMVARS);

if (result < 0)
{
        printf("bc_create_grp error\n");
        dp_status();
        exit(1);
}

/* join the broadcast group */
id_index = bc_join_grp(&id);

if (id_index < 0)
{
        printf("bc_join_grp error\n");
        dp_status();
        exit(1);
}

/* create the remote processes */
sprintf(str_num_equ, "%d", num_equ);
for(i = num_equ; i > 0; i--)
{
        Host_nms[0] = Host_names[(i-1) % NUMHOSTS];
        sprintf(proc_num,"%d",i-1);
        if ((proc_sock[i-1] = dp_create_proc(&host, Host_nms,
                    Proc_nm, str_num_equ, proc_num, 0))<0)
        {
                fprintf(stderr, "dp_create_proc failure\n");
                dp_status();
                exit(1);
        }
        result = dp_write(proc_sock[i-1], (char *) &id, sizeof(id));
```

...main

```
		if (result != sizeof(id))
		{
			printf("write error\n");
			printf("result = %d\n", result);
			dp_status();
			exit(1);
		}
	}

	/* broadcast the data of the group */
	result = bc_send(id_index, var_flags, data);

	/* compute the result X of the equation AX = B */
	proc_ctrler(num_equ, c_ptr, data, var_flags, id_index);

	/* output the results */
	output_rlts(num_equ, c_ptr);

	/* all the processes are terminated so terminate the master */

	/* resign from the broadcast group */
	result = bc_resign_grp(id_index);

	printf("master:result of bc_resign_grp = %d\n", result);

	/* remove the broadcast group */
	result = bc_remove_grp(&id);

	printf("master:result of bc_remove_grp = %d\n", result);

	/* terminate connection with the broadcast server */
	result = bc_close();

	printf("master:result of bc_close = %d\n", result);
}


/*
** proc_ctrler : procedure to exchange data with the remote processes until
**                a solution to the equation AX = b is found
*/
```

proc_ctrler

```
proc_ctrler(num_equ, c_ptr, data, var_flags, id_index)
int	num_equ;
struct	chaos	*c_ptr;
char	data[BUFSIZE];
char	var_flags[NUMVARS];
int	id_index;
{
	int	proc_num;		/* process number			*/
	int	done_flags;		/* when equal to zero a solution has	*/
					/* been found				*/
	int	result;			/* number of bytes received/sent	*/
	int i;


	done_flags = 1;
	while (done_flags > 0)			/* a solution has not yet been found */
	{
		/* set var_flags to receive X and the done_flags */
		for (i = 0; i < NUMVARS; i++)
		{
			var_flags[i] = 0;
		}
```

```
                for (i = 0; i < num_equ; i++)
                {
                        if ( c_ptr->done_flag[i] == 0 )
                        {
                                var_flags[2 * MAXNPROC + i] = 1;
                                var_flags[3 * MAXNPROC + i] = 1;
                        }
                }

                /* wait for remote processes to return results  */
                result = bc_receive(id_index, (short) BLOCK_ON_DATA, data, var_flags);

                /* output the new values of X */
                for (i = 0; i < num_equ; i++)
                {
                        Steps++;

                        fprintf(Ofd, "step number :   %2d          ", Steps);
                        fprintf(Ofd, "          x[%d] = %8.8lf\n", i, c_ptr->X[i]);
                        fflush(Ofd);
                }

                /* determine if all processes are done */
                done_flags = 0;
                for (i = 0; i < num_equ; i++)
                {
                        if (c_ptr->done_flag[i] == 0)
                        /* the remote process i is not done */
                        {
                                done_flags = 1;
                                break;
                        }

                }                       /* end of for */

        }       /* end of while */

}               /* end of proc_ctrler */

/*
** input_data : routine to read data from the file passed in .
*/
```

input_data

```
input_data(argc, argv, num_equ,c_ptr)

int                     argc;
char                    *argv[];
int                     *num_equ;
struct chaos            *c_ptr;

{
        register int    i,j;
        FILE            *data_fp;

        if (argc < 2)
        {
                printf("usage: %s datafile\n", argv[0]);
                exit();
        }

        /* open the input file */
        if ((data_fp = fopen(argv[1], "r")) == NULL)
        {
                printf("cannot open %s\n", argv[1]);
                exit();
        }

        /* read in the number of equations */
        fscanf(data_fp, "%d ", num_equ);

        /* read in the matrix A */
        for ( i=0 ; i < *num_equ ; i++ )
        {
                for ( j=0 ; j < *num_equ ; j ++ )
                {
                        fscanf(data_fp, "%lf",&(c_ptr->A[i][j]));
                }
        }

        /* read in the vector B */
        for ( i = 0 ; i < *num_equ ; i++ )
        {
                fscanf(data_fp, "%lf",&(c_ptr->B[i]));
        }

        /* read in the initial value of the vector X */
        for ( i = 0 ; i < *num_equ ; i++ )
        {
                fscanf(data_fp, "%lf",&(c_ptr->X[i]));
        }

        /* initialize the  done_flags */
        for ( i = 0 ; i < *num_equ ; i++ )
        {
                c_ptr->done_flag[i] = 0;
        }

}

/*
** output_rlts : procedure to output the results
*/
```

output_rlts

```
output_rlts(num_equ, c_ptr)
int     num_equ;
struct  chaos   *c_ptr;
{
        register int i, row, col;

        fprintf(Ofd, "\n\nequation num                  x value\n");
        fprintf(Ofd, "------------                  --------\n\n");

        for (i=0; i < num_equ; i++)
                fprintf(Ofd, "      %d                      %16.12lf\n", i+1, c_ptr->X[i]);

        fprintf(Ofd, "\n\n");
        fflush(Ofd);

        fprintf(Ofd, "program statistics\n");
        fprintf(Ofd, "------- ----------\n\n");

        fprintf(Ofd, "number of hosts                       = %d\n", NUMHOSTS);
        fprintf(Ofd, "number of processes                   = %d\n", num_equ);
        fprintf(Ofd, "total number of steps                 = %d\n", Steps);
        fprintf(Ofd, "degree of precision                   = %lf\n",RLT_PRECISION);
        fprintf(Ofd, "\n");
        fflush(Ofd);
}
```

```
/*
**      BROADCAST VERSION OF THE CHAOTIC RELAXATION  ALGORITHM
**
**      this is the remote process
**
**      remote process to calculate a new x value for the equation sent
**      from the master process
**
*/         Carla Mowers and Isabelle Gerard - June 1986


#include <stdio.h>
#include <signal.h>
#include "/tools/dpup/src/bdct.h"
#include "/tools/dpup/src/dpup_user.h"
#include "/tools/dpup/examples/dgrm/chaos_1.h"

#define  TIMES    2

extern int errno;

main(argc, argv)                                                        main
int argc;
char **argv;
{
        struct chaos *c_ptr;                  /* pointer to the structure          */
                                              /* containing the data of the group */
        char    data[BUFSIZE];                /* buffer to send/receive data of    */
                                              /* the group                           */
        char    var_flags[NUMVARS];           /* flags used to tell which data     */
                                              /* are going to be sent/received     */
        int     sock;                         /* socket to the master process      */
        int     result;                       /* returned number of bytes sent or */
                                              /* received                            */
        int     num_equ;                      /* number of equations                */
        struct  bd_id id;                   /* id of the broadcast group         */
        short   id_index;                     /* index of the current process in   */
                                              /* the broadcast group              */
        FILE    *dfd;                         /* file descriptor for debbuging     */
                                              /* file                                */
        char    dbg_file[50];                 /* name of the debbuging file        */
        short   done_flag = 0;                /* flag set to tell local process    */
                                              /* is done or not                     */
        int     proc_num;                     /* number of the current process     */
        double  sum;                          /* to compute the new value of       */
                                              /* X[proc_num]                         */
        double  diff;                         /* difference between new and        */
                                              /* previous value of X[proc_num]     */
        static  int     counter = 0;          /* consecutive times where the       */
                                              /* difference between the new and    */
                                              /* the previous value of X[proc_num]*/
                                              /* was less than RLT_PRECISION        */
        register short i,j;


        /* set up communication channel with the master process */
        sock = dp_rmt_setup (&argc, argv);

        if (sock < 0)
        {
                printf("remote: dp_rmt_setup error\n");
                dp_status();
                dp_rmt_exit();
        }
```

```
c_ptr = (struct chaos *) data;

/*
** initialize the number of equations passed as an argument by the
**   master
*/
num_equ = atoi(argv[1]);

/*
** initialize number of the current process passed as an argument
** by the master process
*/
proc_num = atoi(argv[2]);


/* read the id of the broadcast group from the master process */
result = dp_read(sock, (char *) &id, sizeof(id), 1);

if (result != sizeof(id))
{
        printf("remote: read error: errno = %d\n", errno);
        dp_rmt_exit();
}

/* establish a connection with the local broadcast server */
result = bc_open();

if (result < 0)
{
        printf("remote: bc_open error\n");
        result = bc_close();
        printf("slave[%d]: bc_close = %d\n",proc_num,result);
        dp_status();
        dp_rmt_exit();
}

/* join the broadcast group */
id_index = bc_join_grp(&id);

if (id_index < 0)
{
        printf("remote: bc_join_grp error\n");
        dp_status();
        result = bc_close();
        printf("slave[%d]: bc_close = %d\n",proc_num,result);
        dp_rmt_exit();
}

/*
** initialize the var_flags to receive A[proc_num][],B[proc_num]
** and X
*/
for (i=0; i < NUMVARS; i++)
{
        var_flags[i] = 0;
}

var_flags[proc_num] = 1;
var_flags[MAXNPROC + proc_num] = 1;
/* receive the initial values of X */
for(j = 0; j < num_equ; j++)
{
        var_flags[2 * MAXNPROC + j] = 1;
}
```

```
/*  receive the data */
result = bc_receive(id_index, (short) BLOCK_ON_DATA, data, var_flags);

done_flag = 0;
c_ptr->done_flag[proc_num] = 0;

while ( done_flag == 0 )
/* loop until done_flag == 1 ( stopping condition is reached ) */
{
        /* compute the new value of X[proc_num] : sum */
        sum = 0.0;
        for ( j = 0 ; j < num_equ ; j++ )
        {
                if ( j != proc_num )
                {
                        sum += c_ptr->A[proc_num][j] *
                                                        c_ptr->X[j];
                }
        }
        sum = (c_ptr->B[proc_num] - sum) / c_ptr->A[proc_num][proc_num];

        /*
        ** check the stopping condition : it is reached, if the
        ** difference between new and previous value of X[proc_num]
        ** is less or equal to RLT_PRECISION  more than TIMES
        ** consecutive times
        */
        diff = sum - c_ptr->X[proc_num];

        if (diff < 0)
        {
                diff *= -1;
        }

        if (diff <= RLT_PRECISION)
        {
                counter++;
                if ( counter == TIMES )
                /* stopping condition reached */
                {
                        done_flag = 1;
                        c_ptr->done_flag[proc_num] = 1;
                }
        }
        else
        {
                counter = 0;
        }

        /* update the X[proc_num] with the new value */
        c_ptr->X[proc_num] = sum;

        /* set the var_flags to send X[proc_num] and the done-flag     */
        for (i=0; i < NUMVARS; i++)
        {
                var_flags[i] = 0;
        }
        var_flags[2 * MAXNPROC + proc_num] = 1;
        var_flags[3 * MAXNPROC + proc_num] = 1;

        /* send X[proc_num] and the done_flag */
        result = bc_send(id_index, var_flags, data);

        if ( done_flag == 1)
```

```
                /* the process is done */
                {
                        /* set the var_flags to send X[proc_num] and */
                        /* the done_flag                               */
                        for (i=0; i < NUMVARS; i++)
                        {
                                var_flags[i] = 0;
                        }
                        var_flags[2 * MAXNPROC + proc_num] = 1;
                        var_flags[3 * MAXNPROC + proc_num] = 1;

                        for ( i = 0; i < 2; i++ )
                        /* send again in case the packet was discarded */
                        /* by the local server of the master           */
                        {
                                /* send X[proc_num] and done_flag[proc_num] */
                                result = bc_send(id_index, var_flags, data);
                        }

                        break;
                }

                else
                /* the process is not done */
                {
                        /*
                        ** initialize the var_flags to receive the new values
                        ** of X ( except X[proc_num] )
                        */
                        for (i=0; i < NUMVARS; i++)
                        {
                                var_flags[i] = 0;
                        }
                        for(j = 0; j < num_equ; j++)
                        {
                                if ( j != proc_num )
                                {
                                        var_flags[2 * MAXNPROC + j] = 1;
                                }
                        }

                        /*  receive the data */
                        result = bc_receive(id_index, (short) ALL_REQ_DATA, data, var_flags);

                }
        }       /* end of while */


        /* resign from the broadcast group */
        result = bc_resign_grp(id_index);

        printf("slave[%d] : result of bc_resign_grp = %d\n",
                        proc_num, result);

        /* terminate connection with the broadcast server */
        result = bc_close();

        printf("slave[%d] : result of bc_close = %d\n",
                        proc_num,result);

        dp_rmt_exit();

}
```

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

MASTER COPY - FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER CS-CU-337-86	2. GOVT ACCESSION NO. N/A	3. RECIPIENT'S CATALOG NUMBER N/A
4. TITLE *(and Subtitle)* DPUP: A Distributed Processing Utilities Package		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* T. Gardner E. Nemeth I. Gerard R. Schnabel C. Mowers		8. CONTRACT OR GRANT NUMBER*(s)* DAAG-29-84-K-0140
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE July 1986 13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

distributed computing; multiprocessing; network of computers

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Attached

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*